Regularization

L2 regularization:

Logistic regression:

$\min_{w,b} J(w,b)$ — loss function. λ = regularization param.

$$\longrightarrow J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$$

L2 regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ — Result will be a scalar.

L1 regularization: $\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}\|w\|_1$.

Neural Network:

$$J(w^{[1]}, b^{[1]}, \cdots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2$$

Frobenius norm:

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(w_{ij}^{[l]})^2$$

$$\rightarrow dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} = w^{[l]} - \alpha\, dw^{[l]}$$

$$\Rightarrow w^{[l]} = w^{[l]} - \alpha\left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$$

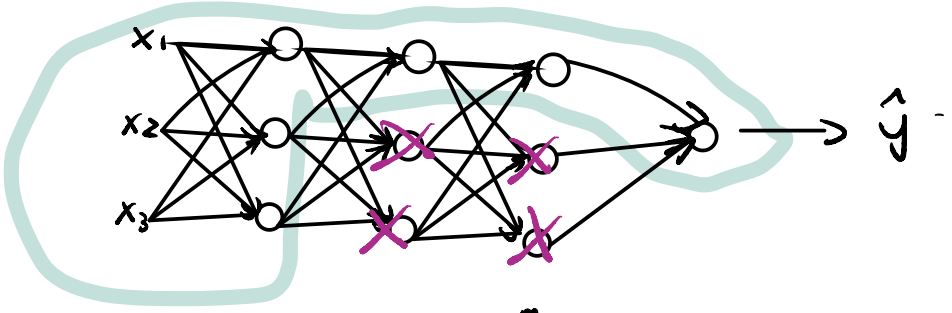
$$= w^{[l]} - \alpha(\text{from backprop}) - \frac{\alpha\lambda}{m} w^{[l]}$$

$$= \left(1 - \frac{\alpha\lambda}{m}\right) w^{[l]} - \alpha(\text{from backprop}).$$

Normally less than 1. -> "weight decay"

Why regularization reduces overfitting

Explaination 1:



$$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

Minimizing J needs $w^{[l]} \approx 0$

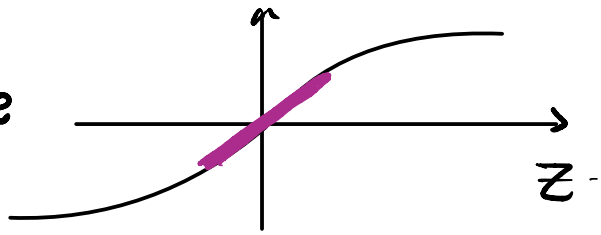
$w^{[l]} \approx 0 \rightarrow$ Some neuron (hidden units)' effects are dimished. if not shut down

Explaination 2:

e.g. tanh: $g(z) = \tanh(z)$.

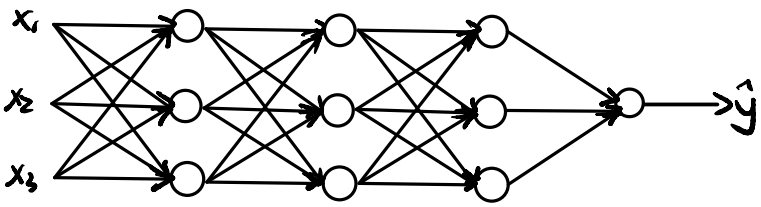
$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

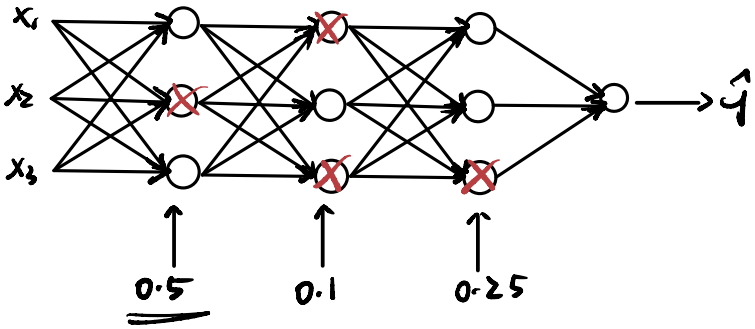
As shown above, $\lambda \uparrow$ $w \downarrow$, narrowing z in to the small range shown in purple line, which makes each layer perform like a linear function. Thus reduce variance. (overfitting).



Dropout regularization.

Intuition: randomly shutting down a small portion of hidden units in each iteration.





possibility of drop any unit from this layer.

Implementation ("inverted dropout").

Illustrate with layer $l=3$, keep-prob $= 0.8$

→ $d_3 = np.random.rand(a_3.shape[0], a_3.shape[1]) < keep\text{-}prob$

$a_3 = np.multiply(a_3, d_3)$. drop out.

$a_3 \mathrel{/}= keep\text{-}prob$.

keep a_3's original magnitude.

(since 20% are shut off, the remaining need to $*0.8$).

→ $(-0.2)x = 0.8x$

$0.8x / 0.8 = x$.

* No dropout when making predictions.

potentially you can dropout multiple times and calculate their mean, using that as predictions for test set.

However, this is very expensive computation which basically produce similar results.

why does drop-out work ?

-> can't rely on any one feature. so have to spread out weights

resulting in a more generalized model, reducing variance.

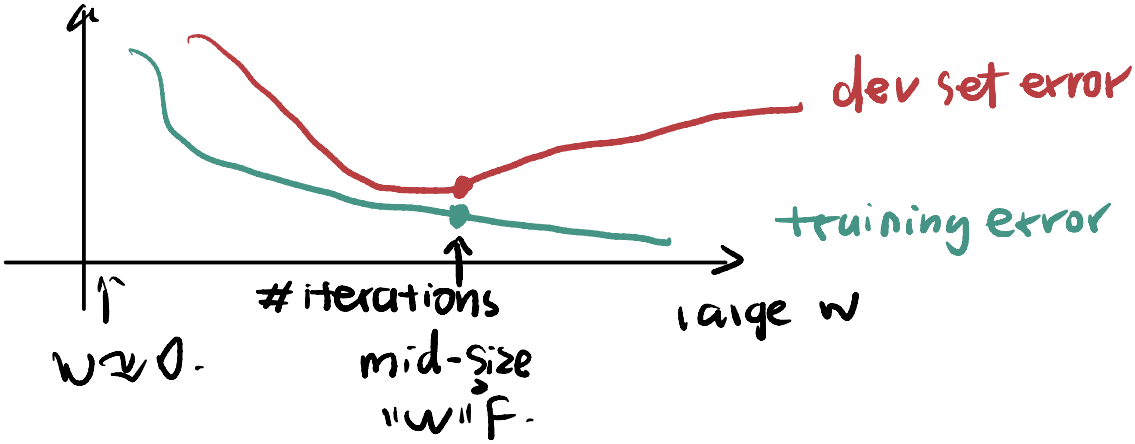
More regularization methods.

Data Augmentation:

e.g. Computer Vision:

through distortion, rotation, revertion, transform pictures in trainingset to increase training size.

Early Stopping



* orthogonalization: achive two goals below one after another.

- Optimize cost function J
  - Gradient Descent ...
  - ...
- Not overfit
  - Regularization ...

early stop can be sometimes problematic because it obeys orthogonalization. Thus, it makes trade-off btw bias and variance.